Electronic Notes in Theoretical Computer Science 72 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume72.html 12 pages

A Unified Framework for Designing, Implementing and Visualizing Distributed Algorithms. [1]

Michel Bauderon and Mohamed Mosbah [2]

LaBRI - University of Bordeaux 1
351 Cours de la Libération
33405 - Talence, France.

Abstract

We present a general method and a toolkit for designing, implementing and visualizing distributed algorithms. We make use of the high level encoding of distributed algorithms as graph rewriting systems. The result is a unified and simple framework for describing, implementing and visualizing a large family of distributed algorithms.

1 Introduction

The process of implementing, debugging, testing and experimenting distributed algorithms is a complex and delicate task, braced with many difficulties and pitfalls. In this context, it is essential to understand the high level algorithmic ideas, independently from the language and platform details of the actual implementations. Visualization environments for distributed algorithms provide tools for abstracting irrelevant program details and for conveying into still or animated images the high level algorithmic behavior of a piece of software. Perception of networks and of events obtained from graphical visualization is most natural to human eyes and conveys more information than streams of texts. The complexity of distributed algorithms, due to interprocess communication and to synchronization, requires conceptual models to simplify and modularize the visualization of distributed algorithms.

The approach we follow in this paper essentially relies on local computations as a basic model to describe distributed algorithms. We present a method

[1] This work is a part of the Visidia project [5], and has been supported by the French CNRS (Centre national de la Recherche scientifique) under the "ATIP Jeunes Chercheurs" funding.
[2] Email: bauderon|mosbah@labri.fr

which automatically produces an implementation of such a distributed algorithm provided that it has been described in terms of local computations. The implementation is then exploited to animate the execution of the algorithm, to help in its validation or to perform experimentations. This is done through a Java toolkit, called Visidia [3,2,4,5] (for Visualization and simulation of distributed algorithms), whose graphical interface allows the user to build a network and to prototype distributed algorithms. Our description of local computations strongly relies on the model of graph relabelling systems as developed by Métivier et al. [10], which provides a strong theoretical basis, many general results and proof techniques. The contribution of this paper is to show that the framework of local computation, which is more general than that of relabelling systems, is useful for studying and visualizing a large family of distributed algorithms. Therefore, we obtain a very general model of our previous work [2] for implementing distributed algorithms.

Consider an anonymous network of processors with arbitrary topology, represented as a connected, undirected graph where vertices denote processors, and edges denote direct communication links. An algorithm is encoded by means of local relabellings. Labels attached to vertices and edges are modified locally, that is on a subgraph of fixed radius k of the given graph, according to certain rules depending on the subgraph only ($k-$local computations). The relabelling is performed until no more transformation is possible. The corresponding configuration is said to be in normal form. Two sequential relabelling steps are said to be independent if they are applied on disjoint subgraphs. In this case they may be applied in any order or even concurrently.

The model of distributed computation is an asynchronous distributed network of processes which communicate by exchanging messages. To overcome the problem of certain nondeterministic distributed algorithms as well as to have efficient and easy implementations, we use randomization [8,15,11]. Métivier et al. [12,13] have investigated randomized algorithms to implement distributed algorithms specified by local computations. Intuitively, each process tries at random to synchronize with one of its neighbours or with all of its neighbours depending on the model we choose, then once synchronized, local computations can be done. A synchronization between two neighbours is called a rendez-vous, and a synchronization between a vertex and all its neighbours is called a star synchronization. Procedures implementing synchronizations are given and discussed in [12,13]. We use these techniques to visualize the execution of a distributed algorithm. All random local synchronizations throughout the network are displayed, and messages exchanged during these synchronizations are also shown. Hence, the visualization of the execution of the whole algorithm is carried out until termination. We have developed a prototype tool with an interactive visual graph editor to build the network, and an interface to implement and visualize distributed algorithms.

The paper is organized as follows. Section 2 recalls local computations and relabelling systems, and their use to describe distributed algorithms. Section 3

presents a general method to automatically produce an implementation of a distributed algorithm encoded by local computations. In Section 4, Visidia, which is a tool based on this model is described. Finally, Section 5 concludes the paper.

2 Theoretical foundations of distributed algorithms

In this section, we illustrate, in an intuitive way, the notion of local computations, and particularly that of graph relabelling systems by showing how some algorithms on networks of processors may be encoded within this framework [9]. As usual, such a network is represented by a graph whose vertices stand for processors and edges for (bidirectional) links between processors. At every time, each vertex and each edge is in some particular state and this state will be encoded by a vertex or edge label. According to its own state and to the states of its neighbours, each vertex may decide to realize an elementary *computation step*. After this step, the states of this vertex, of its neighbours and of the corresponding edges may have changed according to some specific *computation rules*. Let us recall that graph relabelling systems satisfy the following requirements:

(C1) they do not change the underlying graph but only the labelling of its components (edges and/or vertices), the final labelling being the result,

(C2) they are local, that is, each relabelling changes only a connected subgraph of a fixed size in the underlying graph,

(C3) they are locally generated, that is, the applicability condition of the relabelling only depends on the local context of the relabelled subgraph.

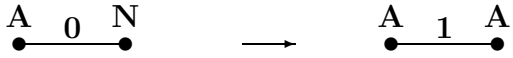
For such systems, the distributed aspect comes from the fact that several relabelling steps can be performed simultaneously on "far enough" subgraphs, giving the same result as a sequential realization of them, in any order. A large family of classical distributed algorithms encoded by graph relabelling systems is given in [2,4]. In order to make the definitions easy to read, we give in the following an example of a graph relabelling system for computing a spanning tree, and an example of local computations for detecting stable properties. Then, the formal definitions of local computations will be presented.

2.1 Distributed computation of a spanning tree

Let us first illustrate graph relabelling systems by considering a simple distributed algorithm which computes a spanning tree of a network. Assume that a unique given processor is in an "active" state (encoded by the label **A**), all other processors being in some "neutral" state (label **N**) and that all links are in some "passive" state (label **0**). The tree initially contains the unique active vertex. At any step of the computation, an active vertex may activate one of its neutral neighbours and mark the corresponding link which gets the

new label **1**. This computation stops as soon as all the processors have been activated. The spanning tree is then obtained by considering all the links with label **1**.

An elementary step in this computation may be depicted as a *relabelling step* by means of the following relabelling rule R which describes the corresponding label modifications (remember that labels describe processor status):

R: 

An application of this relabelling rule on a given graph (or network) consists in (i) finding in the graph a subgraph isomorphic to the left-hand-side of the rule (this subgraph is called the *occurrence* of the rule) and (ii) modifying its labels according to the right-hand-side of the rule.

2.2 Detection of stable properties

The algorithm of Szymanski, Shi and Prywes (SSP's algorithm for short) [16] is a good example to illustrate the notion of local computations.

Consider a distributed algorithm which terminates when all processes reach their local termination conditions, each process is able to determine only its own termination condition. SSP's algorithm detects an instant in which the entire computation is achieved.

Let G be a graph, to each node v is associated a predicate $P(v)$ and an integer $a(v)$. Initially $P(v)$ is false and $a(v)$ is equal to -1. Transformations of the value of $a(v)$ are defined by the following rules.

Each local computation acts on the integer $a(v_0)$ associated to the vertex v_0; the new value of $a(v_0)$ depends on values associated to its neighbours. More precisely, let v_0 be a vertex and let $\{v_1, ..., v_d\}$ the set of vertices adjacent to v_0.
We consider in this section the following assumption. For each node v, the value $P(v)$ eventually becomes true and remains true for ever.

- If $P(v_0) = false$ then $a(v_0) = -1$;
- if $P(v_0) = true$ then $a(v_0) = 1 + Min\{a(v_k) \mid 0 \leq k \leq d\}$.

This algorithm is useful to detect locally the global termination of a distributed algorithm [14,7].
A large family of distributed algorithms can be described as local computations, including election, termination detection, computation of a spanning tree [4]. Let us give now a formal definition of local computations.

2.3 Formal definition of local computations

Local computations are characterized by applications of rules such that: an application of a rule to a ball depends exclusively on the labels appearing in the ball and changes only these labels. The previous examples can be described by the following general model. Let us introduce a few notations. We consider

graphs which are finite, undirected and connected without multiple edges and self-loops. If G is a graph, $V(G)$ denotes the set of vertices and $E(G)$ denotes the set of edges. For a vertex v and a positive integer k; the *ball* of radius k with center v, denoted by $B_G(v,k)$, is the subgraph of G induced by the set of vertices $V' = \{v' \in V \mid d(v,v') \leq k\}$. Let L be an alphabet. A graph labelled over L will be denoted by (G,λ), where $\lambda\colon V(G) \cup E(G) \to L$ is the function labelling vertices and edges. The graph G is called the underlying graph, and the mapping λ is a labelling of G. Let $\mathcal{G}_L$ be the class of graphs labelled over some fixed alphabet L.

Definition 2.1 A graph rewriting relation is a binary relation $\mathcal{R} \subseteq \mathcal{G}_L \times \mathcal{G}_L$ closed under isomorphism. The transitive closure of $\mathcal{R}$ is denoted $\mathcal{R}^*$.
An $\mathcal{R}-$rewriting chain is a sequence $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_n$ such that for every i, $1 \leq i < n$, $\mathbf{G}_i \ \mathcal{R} \ \mathbf{G}_{i+1}$. A sequence of length 1 is called an $\mathcal{R}-$rewriting step (a step for short).

By "closed under isomorphism" we mean that if $\mathbf{G}_1 \simeq \mathbf{G}$ and $\mathbf{G} \ \mathcal{R} \ \mathbf{G}'$, then there exists a labelled graph $\mathbf{G}'_1$ such that $\mathbf{G}_1 \ \mathcal{R} \ \mathbf{G}'_1$ and $\mathbf{G}'_1 \simeq \mathbf{G}'$.

Definition 2.2 Let $\mathcal{R} \subseteq \mathcal{G}_L \times \mathcal{G}_L$ be a graph rewriting relation.

(i) $\mathcal{R}$ is a relabelling relation if whenever two labelled graphs are in relation then their underlying graphs are equal (not only isomorphic):

$$\mathbf{G} \ \mathcal{R} \ \mathbf{H} \quad \Longrightarrow \quad G = H.$$

When $\mathcal{R}$ is a relabelling relation we shall speak about $\mathcal{R}-$relabelling chains (resp. step) instead of $\mathcal{R}-$rewriting chains (resp. step).

(ii) A relabelling relation $\mathcal{R}$ is local if whenever $(G,\lambda) \ \mathcal{R} \ (G,\lambda')$, the labellings λ and λ' only differ on some ball of radius 1 :

$$\exists\ v \in V(G) \text{ such that } \forall\ x \notin V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \lambda'(x).$$

We say that the step changes labels in $B_G(v,1)$.

(iii) An $\mathcal{R}-$normal form of $\mathbf{G} \in \mathcal{G}_L$ is a labelled graph $\mathbf{G}'$ such that $\mathbf{G} \ \mathcal{R}^* \ \mathbf{G}'$, and $\mathbf{G}' \ \mathcal{R} \ \mathbf{G}''$ holds for no $\mathbf{G}''$ in $\mathcal{G}_L$. We say that $\mathcal{R}$ is noetherian if for every graph $\mathbf{G}$ in $\mathcal{G}_L$ there exists no infinite $\mathcal{R}-$relabelling chain starting from $\mathbf{G}$. Thus, if a relabelling relation $\mathcal{R}$ is noetherian, then every labelled graph has an $\mathcal{R}-$normal form.

The next definition states that a local relabelling relation is *locally generated* if its restriction on centered balls of radius 1 determines its computation on any graph.

Definition 2.3 Let $\mathcal{R}$ be a relabelling relation. Then $\mathcal{R}$ is locally generated if the following is satisfied: For any labelled graphs (G,λ), (G,λ'), (H,η), (H,η') and any vertices $v \in V(G)$, $w \in V(H)$ such that the balls $B_G(v,1)$ and

$B_H(w,1)$ are isomorphic via $\varphi\colon V(B_G(v,1)) \longrightarrow V(B_H(w,1))$ and $\varphi(v) = w$, the following three conditions

(i) $\forall\ x \in V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \eta(\varphi(x))$ and $\lambda'(x) = \eta'(\varphi(x))$,

(ii) $\forall\ x \notin V(B_G(v,1)) \cup E(B_G(v,1)),\ \lambda(x) = \lambda'(x)$,

(iii) $\forall\ x \notin V(B_H(w,1)) \cup E(B_H(w,1)),\ \eta(x) = \eta'(x)$,

imply that $(G,\lambda)\ \mathcal{R}\ (G,\lambda')$ if and only if $(H,\eta)\ \mathcal{R}\ (H,\eta')$.

Finally, local computations are the computations defined by a relation locally generated. The reader can find in [4] detailed definitions, formal properties and many examples of local computations.

Let us also note that labels can be sets or sets of sets. In particular, it is possible to handle graphs described as labels. For example, the Mazurkiewicz universal graph reconstruction is a distributed enumeration algorithm which allows the reconstruction of an anonymous graph. The manipulated labels for such an algorithm are sets standing for graphs (see [4]).

3 From relabelling rules to message passing systems

The implementation of a distributed algorithm turns out to be the implementation of its relabelling rules. Therefore, it suffices to implement elementary steps of local relabellings. To do so, we will present the model of the distributed system, then a method to implement local computations.

3.1 Model of the distributed system

The model of the distributed system we shall deal with is a point-to-point network of communicating entities. This system is modelled by a connected simple graph where each node represents an autonomous computing entity (e.g. *thread*, process) and each edge a communication channel. The system is asynchronous; i.e. there is no global clock. The vertices have only local vision of the graph and communicate only with their neighbors by asynchronous messages. More precisely, a vertex v is equipped with ports, numbered from 0 to $(deg(v) - 1)$, which will be used to communicate with neighbours. The attribution of these numbers is completely arbitrary and does not depend on the identities of the neighbouring nodes. We assume that, for a couple of neighbouring vertices, the order of sending messages is the same as that of receiving them. For most algorithms, the network is anonymous which means that vertices have no identities.

3.2 Implementation of local computations (synchronizations)

Although they have in common the fact that the state of a node depends only on the state of (some of) its neighbours, local computations can be of various types, among which the three following types encompass all the algorithms

we have considered so far. We will call a *star*, a vertex together with its neighbours. We refer to these neighbours as the leaves of the star.

RV (Rendez-Vous): in a computation step, the labels attached to a couple of vertices connected by an edge are modified according to some rules depending on the labels appearing on the edge, and on its vertices. The first example presented in Section 2.1 corresponds to this type of computation.

LC_1 (Local Computation of type 1): in a computation step, the label attached to the centre of the star is modified according to some rules depending on the labels of the star (labels of the leaves are not modified). The example presented in Section 2.2 corresponds to this type of computation.

LC_2 (Local Computation of type 2): in a computation step, labels attached to the centre and to the leaves of the star may be modified according to some rules depending on the labels of the star.

Since Angluin [1] has proved that there is no deterministic algorithm to implement local synchronizations in an anonymous network that passes messages asynchronously (see [15]), we use randomized procedures to implement them (see [12,13,2]). Moreover, randomization provides efficient and easy implementations, particularly, in the context of visualization because it allows the user to observe the entire execution of the algorithm as we will show in the sequel.

3.3 Application of relabelling rules

As explained before, each processor tries randomly to get a transient synchronization, with one neighbour or with all its neighbours (depending on the type of local computations). Once a processor v is involved in a synchronization, a rewriting step can be performed. That is, v exchanges its labels and attributes with its neighbour(s), checks if a left-hand side of one of the rules is found, and if so, performs some local computations and updates its labels and its attributes according to the right-hand side of the rule. Then the synchronization is broken. Afterwards, it tries to get another synchronization and so on. Note that many synchronizations can occur at the same time in the network. Since the randomized procedures implementing these synchronizations are Las Vegas [13], each vertex gets involved into a synchronization within finite (and experimentally reasonable) time.

4 Visidia: A tool to prototype and visualize distributed algorithms

We have developed a tool called Visidia [3,2,5] to apply our approach to the visualization and animation of distributed algorithms. It is written in Java where the distributed processors are simulated by the Java threads.

The graphical user interface of the tool, as explained below, is a graphical

environment that allows the user to draw a network easily, and to visualize the execution of a distributed algorithm. The architecture of the tool is composed of three main parts, namely the graphical user interface, the simulator and the algorithm library as sketched in **Fig.1**. These modules are well-separated such that the modification of one component involves only small changes for the rest of the system. The ongoing work will provide Visidia with a network interface for the visualization of real-world distributed systems, which will be available in a later version of our tool.

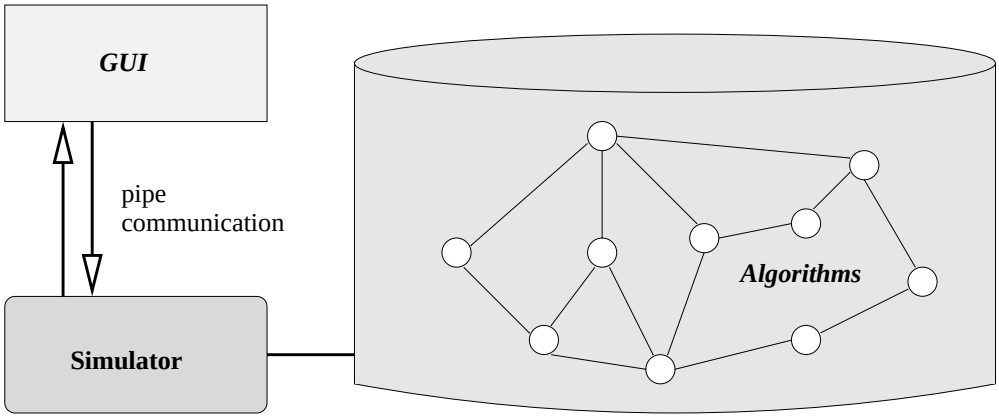


Fig. 1. *architecture of the prototype*

4.1 The Graphical User Interface

Visidia's GUI produces an editor allowing the user to construct a network by "Drag & Drop". The user can add, delete, or select vertices, edges or subgraphs. The visual attributes of a vertex —labels, colors, and shapes— may also be set by the user. Once the network is drawn, the simulation is run after the user has chosen an algorithm. During runtime, the traffic of messages exchanged between nodes and their values are displayed, and the status of edges and nodes are updated on-the-fly. Moreover, the visualization speed can be chosen by the user such that the messages can be seen travelling slowly or fast, depending on the purpose of the current application.

4.2 The Simulator

The simulator is the link between the visualizer and the algorithms. It models a network of asynchronous processors. Each processor communicates only with its immediate neighbours by message passing. In the current version of the tool, each processor, associated to a vertex in the underlying graph, is implemented by a JAVA *thread* as mentioned above. The simulator manages the exchanges of messages between threads, as well as the visualization of events. Each event has an identifier which is a number that will be used to synchronize the execution of the algorithm with its visualization. The identifier of an event helps to acknowledge its visualization. In this way,

displayed information is synchronized with the state of the network. There are two types of events corresponding to the state modification of vertices or edges, or to message exchanges:

- **State modification** of a vertex: The state of a vertex is a set of informations which are used by the algorithm during its execution. These informations are interpreted by the graphical interface. Precisely, when a vertex changes its state, it informs the simulator who sends an event to the graphical interface. The process of this vertex waits until obtaining an acknowledgment receipt from the graphical interface, which means that this event has been displayed. The changes of an edge are displayed in a similar way.
- **Message visualization**: Processors use queues to store messages. Sending a message M from vertex A to B consists of adding M to the queue of B. However, the visualization of the message M moving from A to B is displayed before adding it effectively to the queue of B. The simulator begins by sending an event to the graphical interface in order to visualize the message M. Once this event is acknowledged by the simulator, it is added to the queue of B, and a signal is sent to the processor of B in order to wake it in case it had been blocked on reading an empty queue.

4.3 The Algorithm Library

An algorithm is implemented by a JAVA program which will be instantiated on each vertex of the graph, and executed asynchronously by the corresponding processor. A vertex is implemented by a class that contains its identifier, its internal state, its degree, and optionally the size of the graph. It is possible for the programmer to manipulate a vertex by using the following implemented interface functions:

- `rendezVous()`: a function that returns the neighbour with whom the synchronization occurs.
- `starSynchro1()`: returns the center of the star during a star synchronization. Only the center can update its attributes.
- `starSynchro2()`: returns the center of the star during a star synchronization. The center and its neighbours can update their attributes.
- `breakSynchro()`: Stops the synchronization. Vertices involved in the current synchronization are no longer locked, they try again to catch other synchronizations.
- `getId()`: returns the identity of a vertex (for networks with processor identities),
- `getState()`: (resp. `setState()`) gives (resp. changes) the state of a vertex,
- `getArity()`: returns the degree of a vertex, i.e. the number of neighbours,

- `getNetSize()`: allows to know the size of the graph for algorithms whenever the size is assumed to be known.

Since communications between processors are based on messages, the required functions to handle messages are provided. A message is programmed by a class that contains all required informations. The programmer can use a message through the following methods:

- `sendTo(int)`: (resp. `sendAll()`) sends the message to a particular neighbour (resp. all neighbours),
- `receiveFrom(int)`: (resp. `receive()`) receives a message from a particular neighbour (resp. the first message waiting in the queue of the vertex). Other methods are also available to manage messages of particular types, i.e. messages that send integer, strings, or other types.
  Remember that although messages are stored in the queue of the receiver, the implementation allows also to handle messages that arrive on a particular channel.

Many distributed algorithms described by local computations have already been implemented and can be directly animated [4,5]. These include the following :

- leader election in trees, in chordal graphs and in complete graphs,
- randomized Rendez-vous and randomized local elections,
- spanning tree in anonymous networks,
- spanning tree in networks with identities,
- Mazurkiewicz's universal graph reconstruction,
- detection of stable properties,
- Chang-Robert's algorithm,
- 3-coloration of a ring,
- Dijkstra-Scholten's termination detection algorithm,
- Dijkstra-Feijen-Van Gasteren's termination detection algorithm,
- Ricart-Agrawala's mutual exclusion algorithm.

An interesting advantage of our approach is that we only need to implement local rewritings to code complicated distributed algorithms. Therefore, visualizing the execution of these algorithms consists of animating distributed local computations.

5 Conclusion

Visidia is a tool for the execution and visualization of distributed algorithms. It is motivated by the important theoretical results on the use of graph local computations to encode distributed algorithms. The result is a high-level encoding of distributed algorithms by means of relabelling rules. The Visidia

interface allows the user to edit a netwok and to animate the execution of a distributed algorithm. Visidia can also be executed on a network of distinct machines. In this case, the vertices of the graph, and hence the threads, may be located on different machines. We have used the Java RMI library to extend the distribution of Visidia. Moreover, we have recently added the ability to deal with failures. The user can simulate a failure of a processor or a link by using the graphical interface of Visidia, and can handle such a failure through some provided primitives. We are currently working on adding reliability to the distributed algorithms we have studied. Our aim is to make the survival of the execution of a distributed algorithm in spite of failures. We think that this problem can be solved by using relabelling rules, and that a distributed algorithm encoded by local computations can be transformed an equivalent reliable one. Finally, let us mention that Visidia is useful for students to understand distributed computing since they can observe the execution of distributed algorithms.

References

[1] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Symposium on theory of computing*, pages 82–93, 1980.

[2] M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. In *Second International Workshop on Graph Transformation and Visual Modeling Techniques, Crete, Greece*, July 12-13, 2001.

[3] M. Bauderon, S. Gruner, and M. Mosbah. A new tool for the simulation and visualization of distributed algorithms. Technical Report 1245-00, LaBRI, 2000. Accepted in MFI'01, Toulouse, 21-23 May 2001.

[4] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. Technical Report RR-1271-02, LaBRI, 2002.

[5] M. Bauderon, M. Mosbah, and A. Sellami. Visidia: A tool for the visualization and similation of distributed algorithms. *http://www.labri.fr/visidia/*.

[6] Hartmut Ehrig, Kreowski Hans-Jörg, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific, 1999.

[7] E. Godard, Y. Métivier, M. Mosbah, and A. Sellami. Termination detection of distributed algorithms by graph relabelling systems. In 2002 LNCS, editor, *First International Conference on Graph Transformation Barcelona (Spain)*. Springer-Verlag, October 7-12, 2002.

[8] R. Gupta, S. A. Smolka, and S. Bhaskar. On randomization in sequential and distributed algorithms. *ACM Comput. sur.*, 26(1):7–86, 1994.

[9] I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabelling systems. *Math. Syst. Theory*, 28:41–65, 1995.

[10] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H.J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.

[11] J. Ramirez-Alfonsin M. Habib, C. McDiarmid and B. Reed, editors. *Probabilistic Methods for Algorithmic Discrete Mathematic.* Springer-Verlag, 1998.

[12] Y. Métivier, N. Saheb, and A. Zemmari. Randomized rendezvous. In *Mathematics and computer science : Algorithms, trees, combinatorics and probabilities*, Trends in mathematics, pages 183–194. Birkhäuser, 2000.

[13] Y. Métivier, N. Saheb, and A. Zemmari. Randomized local elections. *Inform. Proc. Letters*, 82:313–120, 2002.

[14] Y. Métivier and G. Tel. Termination detection and universal graph reconstruction. In *International Colloquium on structural information and communication complexity*, pages 237–251. Carleton scientific press, 2000.

[15] G. Tel. *Introduction to distributed algorithms.* Cambridge University Press, 2000.

[16] B. Szymanski Y. Shi and N. Prywes. Terminating iterative solutions of simultaneous equations in distributed message passing systems. In *4th International Conference on Distributed Computing Systems*, pages 287–292, 1985.